Université de Nice - Sophia Antipolis

Polytech Nice Sophia

Sciences Informatiques 5ème année

SOA

Rapport de projet : Blue Galactic X

Equipe J

David BISEGNA, Jason HAENLIN, Yassine JRAD, Betsara MARCELLIN

Enseignants: P.Collet, L.Gaillard, M.Chevalier

Table des matières

Notre vision du sujet **2**

Acteurs 2

Choix 2

Décision sur l'architecture **4**

Interfaces de communications 4

gRPC en quelques points 5

Stack techniques 6

Spring boot en quelques points 6

Sérialisation / Désérialisation 6

Architecture Actuelle **8**

Présentation 8

Rocket Web Service 8

Mission Web Service 9

Telemetry Web Service 9

Weather Web Service 9

Payload Web Service 9

Faiblesses 9

Rocket Web Service 9

Mission Web Service 9

Telemetry Web Service 10

Payload Web Service 10

Globalement 10

Reste à faire **10**

Difficultés rencontrées 10

Implémentation Envisagé 11

Evolution de l'architecture 13

Prise de recul **13**

Rétrospective **14**

Conclusion **14**

Notre vision du sujet

Acteurs

Le sujet fait part de plusieurs acteurs, 6 en tout. Nous avons également un acteur Scheduler qui va nous permettre de mettre à jour les données de télémétrie. Les acteurs qui interagissent avec notre système sont donc les suivants :

- Richard : Le commandant de mission qui va se préoccuper du bon déroulement de la mission, en exploitant les informations qui lui sont transmisent par les autres départements.
- Elon : Le chef du département Rocket qui va être responsable de l'état de la fusée jusqu'au départ de cette dernière puis va pouvoir effectuer des commandes sur la fusée, pendant la mission.
- Tory : L'officier du département Weather s'occupe de rendre compte des conditions météorologiques au commandant de mission.
- Jeff : L'officier du département Telemetry récupère et enregistre les informations de la fusée, de ses boosters et du payload pendant toute la durée de la mission.
- Gwynne : Le chef du département Payload consulte les informations de la charge (payload) transporté pour savoir si ce dernier a atteint et resté sur le bon orbit.
- Peter : Le PDG de Blue Origin X veut savoir si le booster est revenu à la base ou non.
- Un "Scheduler" qui s'occupe d'ajouter une notion de temps, ce qui nous permet de pouvoir réellement visualiser les données télémétriques qui évoluent dans le temps, et ainsi avoir des comportements automatiques en fonction de ces métriques (comme par exemple la séparation entre la fusée et les boosters lorsqu'on atteint un certain niveau d'essence). Sans ça, nous serions obligés de changer l'état de tous nos composants instantanément entre chaque actions.

Ce sont tous des acteurs qui vont interagir pendant toute ou une partie de la durée de la mission. On notera que le chef du département Rocket peut ordonner le lancement de la fusée, séparer les boosters de la fusée, mais n'aura pas à gérer la puissance des booster pendant le passage au travers de la zone Max Q.

Choix

Pour le premier set d'user-story, nous avions compris que le GO/NO GO poll ne nécessitait pas l'intervention des personas (Richard, Elon, Tory). Richard ne pouvait qu'autoriser Elon à effectuer le lancement de la fusée.

Plus tard, nous avons pensé à une vision plus détaillée et "réaliste" de notre ancien système. Les responsables des départements weather, rocket et mission pouvaient générer un rapport qui est consultable par les autres départements qui en ont besoin. Ce rapport permet aux autres départements de prendre des décisions, un scénario d'exemple serait que : Tory consulte la météo, elle génère un rapport dans lequel on retrouve les données météorologiques. Elon, lui, récupère les informations sur la fusée puis génère un rapport avec les données concernant la fusée. Richard lui a accès à ces rapports, il récupère donc le rapport du département Weather, puis Rocket, ensuite il peut donner l'autorisation au département Rocket de lancer la fusée, si Richard valide, alors Elon lance la fusée.

Pour monitorer la fusée et sa destination, nous avons pensé qu'il aurait été intéressant d'implémenter un système de coordonnées similaire à ceux utilisé pour les satellites gravitant autour de la terre (latitude, longitude et altitude). Au final, nous avons prévu que la mission pourrait prendre plus d'ampleur, jusqu'à effectuer la gestion des voyages interplanétaires, ce qui rendrait désuet le système de coordonnées (latitude, longitude, altitude). Nous avons alors le choix d'implémenter un système de coordonnées céleste ou d'implémenter un système de coordonnées sur 3 dimensions dont l'origine serait la terre, tout cela avec l'hypothèse que les objets célestes ont une position fixe dans le temps par rapport à ce référentiel. Avec notre expérience du deuxième sprint (nous avons voulu donner beaucoup de détails pour se rapprocher d'un scénario réel, ce qui nous a causé du retard pour un gain non conséquent), nous avons choisi de simplifier ce système de coordonnées, car dans une optique purement MVP, un système de coordonnées en 3D "géocentrique" est tout aussi pertinent. Si nous devions faire évoluer ce modèle plus tard, il suffirait juste de mettre à jour l'entité SpaceCoordinate et les méthodes des interfaces qui gèrent le déplacement de la fusée et ses composants (booster, payload).

Pour répondre au besoin de Richard qui est de détruire la fusée en cas d'anomalie, un appel RPC a été ajouté au service de Rocket pour détruire celle-ci. De plus, nous avons ajouté au service de telemetry un appel pour connaître des anomalies liées aux données télémétriques de la fusée ou des boosters (Dans le futur nous pourrions ajouter des alertes automatiques pour générer des anomalies si certaines données dépasse un seuil donné). Richard peut donc soit détruire la fusée si une anomalie justifie cela, ou bien en visualisant les données télémétriques brute et en concluant lui-même qu'il y a une anomalie.

Décision sur l'architecture

Avant de nous lancer dans le projet, nous avions pris le temps de comparer certaines stack technologiques tel que NodeJs et Spring-boot. Qui plus est, il fallait bien prendre connaissance des différents types d'interfaces et de styles de programmations.

Interfaces de communications

Nous comparons ici le REST avec le gRPC et le SOAP. Le gRPC est une interface spécifique, nous voulons donc le comparer en tout premier par rapport à nos autres choix.

Interface	REST	gRPC	SOAP
Données ciblés	Ressources	Action	Action
HTTP protocol	HTTP/1.1,2	HTTP/2 (proxy si 1.1)	HTTP/1.1,2
Type du payload	Json	Binaire	XML
Taille du payload	Json compacte	moindre que les autres	Très verbeux
Force de typage	Nombre, string, list, bool	Typage fin (eg. int16)	Typage fort
Contrat fort ?	non	oui	oui
Interface Lang. Desc.	tiers : swagger	Proto file	WSDL
Génération de code	Possible avec swagger depuis le code	A partir du contrat	A partir du contrat
But principal	Accéder à des ressources	Optimiser les échanges en réseau (mieux pour le Service To Service)	faire une action sur le serveur
Facilité d'utilisation	Le plus simple	Configurer les dépendances mais facile avec le framework	Beaucoup de configuration et problématique si Contrat first
Outil de test disponible	Navigateur/Postman	BloomRPC	SoapUI

Nous avons aussi comparé d'autre type de RPC ou Document (encodage littéral) sur certains points.
XML-RPC et XML-DOC qui sont des interfaces SOAP et JSON-RPC qui est la deuxième version de RPC sortie plus tard que XML-RPC.

Interface	XML-RPC	JSON-RPC	XML-DOC	SOAP
Taille du payload	Plus concis pour un XML	Un Json avec des spécifications à inclure (type, version, id)	Peut être validé dans un validateur XML, WSDL moins lisible	Comme XML-DOC mais plus lourd encore
Force de typage	Que des types simples	Nombre, string, list, bool	Typage fort	Typage fort
Interface Lang. Desc.	WSDL	AUCUN	WSDL	WSDL
Facilité d'utilisation	nécessitent beaucoup de configuration (WSDL ou POJO)	Un objet Json avec un Endpoint REST Suffi	nécessitent beaucoup de configuration	nécessitent beaucoup de configuration

Pour le projet, étant donné qu'il faut pouvoir gérer des requêtes pour accéder aux ressources et d'autres pour émettre des actions, nous avons pris REST ainsi que gRPC pour nos actions. Pour le gRPC notre choix s'est porté sur la simplicité, la réutilisabilité et la performance. La définition du contrat sous forme de Protocol Buffer permet de générer du code dans de multiple langages. Nous savons que nos Web Service doivent être indépendants et qu'il est possible d'avoir des langages différents. Avoir un contrat fort compréhensible par de multiple langages est essentiel.

gRPC en quelques points

- Description de l'API exposée
- Rétrocompatibilité grâce à des indexes, tous les champs peuvent être optionnels. Donc on peut avoir la nouvelle version mais garder l'ancienne version lorsque l'on est en cours de migration.
- Optimisation des échanges réseaux (binaire plus petit)

- Génération de code client/serveur dans beaucoup de langage avec protoc
- Type fort et fin
- Protocol Buffers comme Langage de description
- Uniquement HTTP/2 qui est de toute façon plus performant que HTTP/1.1 mais un proxy-web est disponible pour les navigateurs

Stack techniques

Pour la techno à utiliser. Nous avions débattu entre Nodejs, Spring Boot et Java EE (choix personnels). Nous avons opté pour Spring boot, nous savons pertinemment que Nodejs est bien adapté pour la scalabilité et la programmation événementielle, néanmoins, en Nodejs il est très difficile de structurer le code convenable, nous pouvons le faire mais c'est beaucoup plus compliqué à maintenir que Spring Boot. Même si la JVM a aussi une contrepartie et est plus coûteuse qu'une application NodeJs. Nous aurions voulu opté pour Deno, une alternative à Node mais elle est encore trop récente et manque de dépendance (gRPC, Kafka, etc.).

Spring boot en quelques points

- Simple d'intégration pour les tests / base de données / etc.
- Marshalling et Unmarshalling intégré
- Langage objet très fort (plus apte pour du DDD)
- Maven pour gérer les dépendances, les versions et le build
- Documentation hallucinante et bien faite
- Attention : Synchrone par défaut par rapport à NodeJs qui est Evenementiel

Sérialisation / Désérialisation

Nous avons aussi comparé la sérialisation et désérialisation. La vitesse des transmissions est importante d'où l'importance de la taille de la requête. Néanmoins, la sérialisation et la désérialisation est très importante et peut être très onéreuse.

Format	**XML**	**Json**	**Protobuf**
Briefness	Très verbeux	syntaxe concise avec clé:valeur	le format binaire, le plus concis de tous
Human Readability	Lisible mais avec beaucoup de tags	facile à lire	non lisible par l'homme
Performance	rapide mais	le format lisible par	format très très

	généralement plus lent que Json	l'homme le plus rapide	rapide

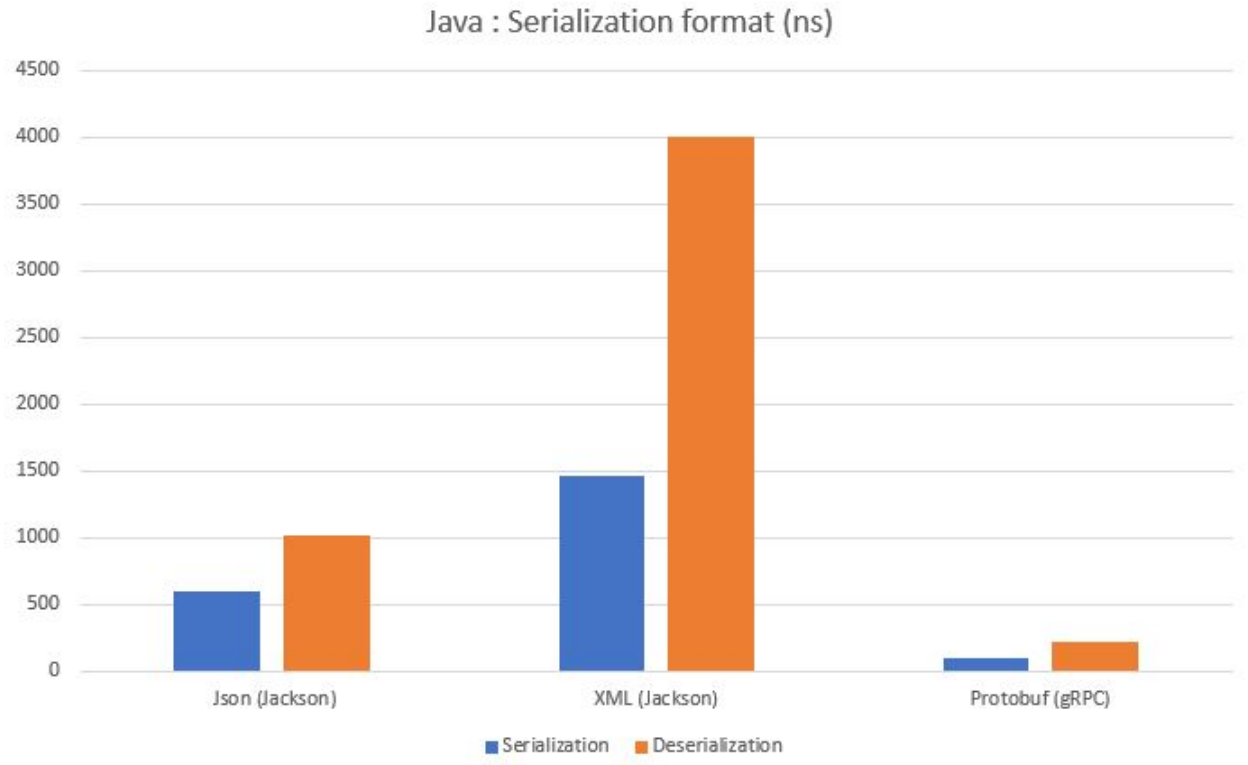
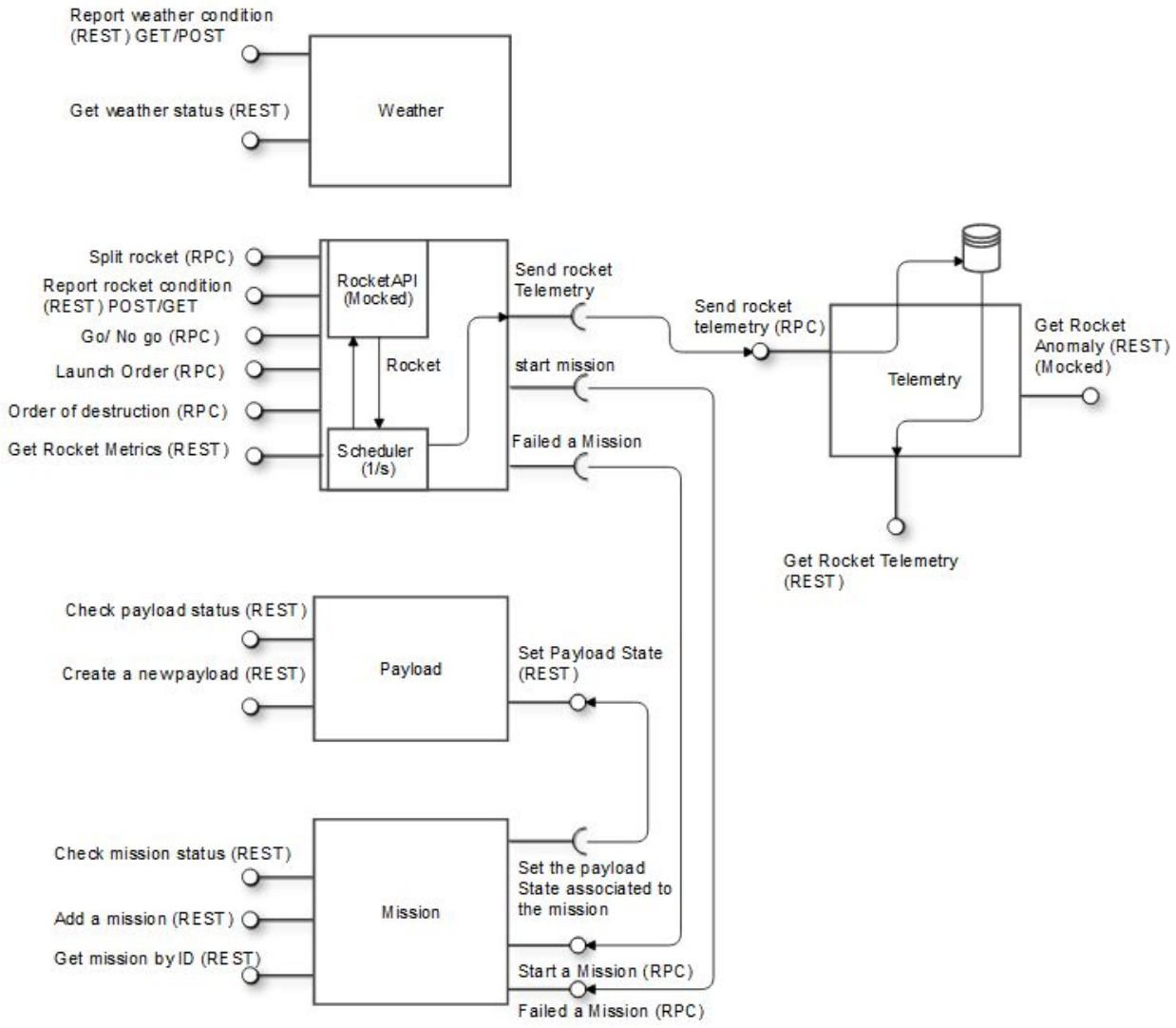


Figure 1 : performance de sérialisation / désérialisation

Il faut prendre les résultats comme valide principalement pour Java car selon les langages, les performances peuvent radicalement varier. Nous pouvons remarquer sur la figure 1 que le protobuf est loin devant en terme d'encodage et décodage. Ces résultats nous conforte dans notre choix et l'utiliser pour kafka par la suite.

Architecture Actuelle

Présentation



Rocket Web Service

C'est le web service qui gère les Fusées. Il s'occupe des communications entre la fusée Physique et le serveur (lui-même). Tout ordre de lancement, destruction ou adaptation de la rocket passe par ce service. Un scheduler est aussi présent pour récupérer périodiquement les mesures des capteurs sur la fusée pour les transmettres aux autres services s'ils en ont besoin afin qu'ils puissent les utiliser pour d'autres tâches (ex : télémétrie, anomalie).

Mission Web Service

Ce service défini ce qu'est une mission. Chaque mission est associé à une rocket, un payload et un objectif (coordonnée). Ce service nous permet de gérer les missions qui sont créées et de les mettres à jour durant toutes les étapes de celle-ci.

Telemetry Web Service

Il gère les métriques qui arrivent de rocket. Il est aussi chargé de la persistance, d'en déterminer les anomalies et de transmettre les données.

Weather Web Service

Vérifie la météo actuelle et permet à un météorologiste d'écrire un rapport de ses observations.

Payload Web Service

Ce service défini ce qu'est un payload. Chaque payload est associé à une rocket, un payload et un objectif (coordonnée)

Faiblesses

Rocket Web Service

Nous pensons qu'il y a certaines incohérences dans certaines entités. Si nous prenons Rocket, c'est l'agrégat principale qui contient les metrics que l'on utilise pour récupérer les informations relatives à la Rocket avant de faire le Go/NoGo de la mission. Néanmoins, Quand on relève les informations de la requête périodiquement pour les telemetry pendant le déroulement de la mission, RocketAPI (mocked) nous renvoie des métriques avec les informations d'une rocket, mais de ce fait, selon l'étape du scénario, cela varie, il faudrait revoir cette partie pour avoir une meilleure cohérence à chaque étape. Il faudrait certainement avoir tout dans notre agrégat, il faudrait donc mettre à jour seulement cet agrégat dès qu'une information est mise à jour.

Mission Web Service

Une partie qui semble à revoir est la mise à jour du statut des différentes entités (Rocket, Mission et Payload). Quand la mission démarre, Rocket Web Service fait une Action à Mission Web Service pour dire que la mission à commencer et mettre à jour le statut. Néanmoins, il va à son tour mettre à jour le statut de Payload Web Service car ils ont tous un Statut (Sur la rocket, le payload et la mission). Comme c'est une requête REST (Mission -> REST -> Payload), il y a une duplication de PayloadStatus dans Mission WS. Il nous semble très compliqué de le maintenir sur le long terme si on est trop couplé à un autre Web Service, il faudrait un moyen de mettre à jour tous les statuts de tous les WS avec une grammaire similaire (ils doivent se

comprendre ou du moins traduire à partir d'une grammaire qu'ils comprennent entre eux). Ca nous semble contraignant à l'heure actuelle.

Telemetry Web Service

Concernant la persistance, nous stockons les dernières données de la fusée. Si des nouvelles fonctionnalités impliquent le traitement de toute une mission (par exemple), il faudra garder l'historique afin d'en ressortir des statistiques (par exemple).

Payload Web Service

Le métier lié à un payload reste limité pour l'instant. Pour le moment, il a plus le comportement d'une entité plutôt qu'un vrai métier.

Globalement

- Duplication des objets métier dans les différents services
- Rocket Web Service qui commence à être très gros
- Pour les communications inter web service, l'émetteur doit connaître le destinataire et le contrat de la requête, couplage entre les deux car si l'un change, l'autre ne pourra plus fonctionner correctement.
- Manque d'indépendances, si un service est down, il n'aura pas la possibilité de lire les dernières requêtes quand il sera rétabli car il y a une seule transmission. Ou sinon il faut gérer les cas d'erreur dans tous les expéditeurs.
- Si l'on souhaite envoyer des informations à plusieurs WS, il faut faire plus requêtes.

Reste à faire

Nous avions rencontré des difficultés durant les premières semaines ainsi que des mauvais choix nous ont coûté beaucoup de dettes techniques du au rework à faire.
En conséquence du retard lié à nos choix d'implémentations, nous n'avons pas implémenté les user-stories 9, 10, 11, et 12.

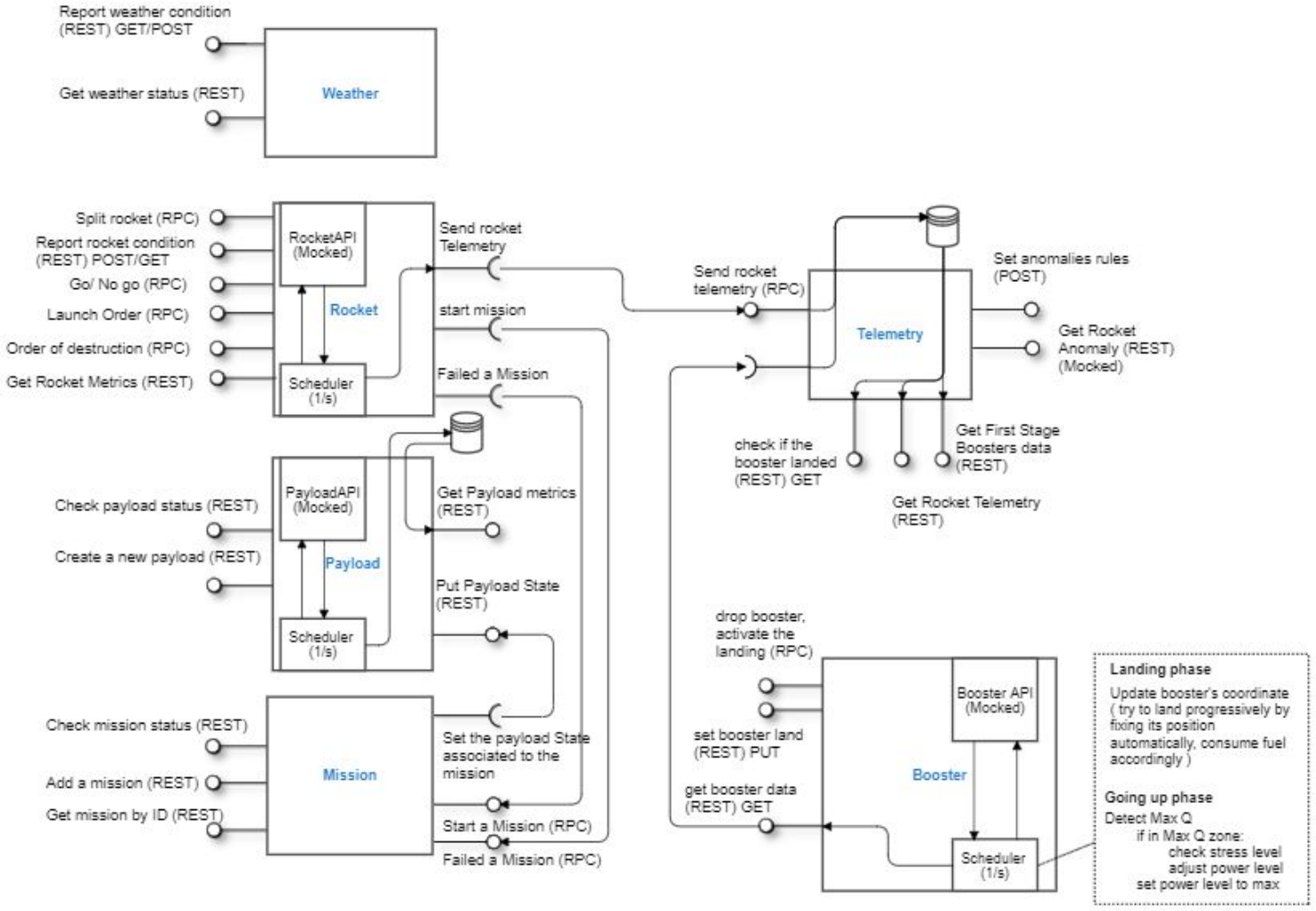
Difficultés rencontrées

Nous avions eu des problèmes de compréhension et avions mal interprété certains aspects du projet et des commentaires laissés par les enseignants.
Sur la première vague de User Story, nous avions mis la possibilité de récupérer des données pour la météo et la rocket, cependant Richard pour vérifier la rocket et la météo avait les mêmes actions à faire que Tory et Elon. Nous nous sommes rendu compte de cette incohérence un peu tard. Nous avions réfléchi à une meilleure implémentation pour que Elon et Tory puisse avoir un

impact (une utilité) dans le système. Nous avions donc ajouté la possibilité de générer un rapport pour la météo et la rocket donnant un métier plus concret à Elon et Tory. Richard quand à lui devait voir ces rapports pour vérifier si tout était prêt avant de faire un Go/No Go. Néanmoins, on nous avait dit qu'il fallait être un minimum concret, mais nous avions cherché à être "trop" concret et avions perdu beaucoup de temps.

Implémentation Envisagé

Au cours du prochain sprint, nous ajouterons un nouveau service "Booster", ce service s'occupera de la phase montante et descendante du Booster. Ci-dessous notre schéma d'architecture prévisionnel pour l'implémentation des US 9,10,11,12 :



Le webservice booster va gérer automatiquement la puissance des moteurs et la direction des booster. Une API mocké représentera le booster physique réel, elle simulera la consommation de fuel selon la puissance demandée, le niveau de stress lié à la vitesse et l'altitude et également la position du booster physique selon la commande de direction envoyée par le webservice booster, ce dernier simulera aussi la chute du booster (retour à la station). La phase de montée et de descente (post détachement) sont gérés automatiquement par le webservice booster. En effet, nous avons décidé que ces actions doivent se faire automatiquement par le système, car dans ces scénarios, un système informatique aura une meilleur réactivité qu'un humain pour effectuer les ajustements de puissance et direction au bon moment. Malgré tout, il peut toutefois être pertinent d'implémenter une commande manuelle dans le cas où le système présente un problème pendant le vol, mais cela ne pose pas de problème car il suffirait d'ajouter de nouvelle routes/méthodes à l'interface exposée par le webservice booster. Grâce à un scheduler qui fait un appel par seconde :

- Lors de la phase de montée, le scheduler du webservice booster demande la position du booster à l'API mocké, si elle a atteint la zone de Max Q (une distance par rapport à la station), le scheduler demande à nouveau à l'API mocké le niveau de stress de la fusée, si ce niveau de stress dépasse un seuil, le webservice booster réduit la puissance du moteur proportionnellement au dépassement du seuil. Une fois la zone de Max Q dépassé, les moteurs sont remis à 100% de leur puissance.
- Lors de la phase de descente, le scheduler demande la position de la fusée, ensuite le webservice appelle une méthode interne qui compare la position récupérée ainsi que la position de la station et renvoie les commandes d'ajustement de position à l'API mocké (ces commandes pourrait consommer du carburant par exemple si on doit aussi gérer l'amortissement de l'atterrissage). Cela est réitéré jusqu'à ce que la fusée arrive à destination ou a eu un problème pendant la descente.
- Les données relatives au booster sont envoyées périodiquement au webservice Telemetry

Avec cette architecture, nous répondrons donc à l'US 9 via son interface Peter pourra faire un appel GET (`boolean *didTheBoosterLand(String boosterId)*`) au service télémétrie pour savoir si le booster a réussi à atterrir, ce GET va récupérer l'information d'atterrissage du booster dans la base de données. Pour l'US 10, à l'aide de son interface, Jeff fera un appel GET (`Booster retrieveTheBoosterDatas*(String boosterId)*`) vers le webservice Booster pour voir les dernières informations relatives au booster à tout moment. Concernant l'US 11, Gwyne en utilisant son interface pourra faire un appel GET (`Payload retrievePayloadMetrics(String payloadId)`) au webservice Payload pour récupérer l'ensemble des informations disponibles du payload. Pour répondre à l'US 12, le webservice Booster fera une gestion automatique de ce passage à travers Max Q.

Prise de recul

Sur ce premier set de scénarios, nous remarquons des difficultés à mettre en place certain aspect des web services. Nous constatons que dès lors qu’une communication doit être faite entre deux services, un fort couplage commence à apparaître. Nous pensons qu’avec l’architecture orientée événements, en utilisant Kafka, nous pourrons réduire fortement le couplage, notamment vis à vis du service de telemetry car il pourra simplement envoyer les données de télémétrie dans un bus, sans devoir lier les services entre eux.

- Besoin de connaître le EndPoint du serveur avec lequel on veut communiquer
- Élasticité verticale possible car le service garde le même Endpoint
- Élasticité horizontale compliqué car ça implique un nouveau Endpoint qu’il faut découvrir. Plus un load Balancing compliqué.
- Approche actuelle pas scalable horizontalement
- Envoyer un message à plusieurs web service implique plusieurs requêtes.

Au niveau de notre implémentation. Certains points sont à revoir car inconsistant dans nos web services. Hormi les requêtes entre services, il y a aussi beaucoup de requêtes déclenchés manuellements. Nous supposons que pouvoir détacher l’étage ou détecter les anomalies devraient être déclenché automatique grâce à des règles définies par un acteur (ex : si gyroscope de la fusée à -90° (vers la terre) alors le système crée une anomalie automatiquement).

Nous sommes aussi conscient que certains web services sont pour le moment très vide avec un métier encore compliqué à définir. Par manque de temps, nous avons décidé de garder notre architecture actuelle. Nous rattraperons petit à petit notre retard et allons retravailler les web services par rapport aux nouveaux scénarios en espérant qu’il nous permettront de mieux clarifier nos interrogations.

Rétrospective

Au départ, nous avions pris de mauvaises décisions mais nous nous efforçons d'avoir un projet rigoureux que ce soit dans la technique ou la méthode de travail.

Mauvaises Décisions	**Actions**
Avoir fait des submodules	Bien vérifier les consignes ou demander une confirmation si incertain.
Mauvaises premières implémentations de Weather et Rocket Service	Mieux prendre en considération les métiers de chaque acteur. Richard ne devait pas faire la même chose que Tory ou Elon.
Trop de retard pour retravailler au lieu d'apporter des fonctionnalités	Même si on nous demande d'être concret, faire le minimum, on est parti à l'extrême dans les données
US trop grosse parfois (sur plusieurs ws)	Chaque Web service sont indépendants. Une US doit toucher qu'un Web service à la fois
Manque de traçabilité des anciens choix d'architecture	Pendant les réunions (ou après), indiquer dans un doc ou wiki du projet les idées qui nous sont venus pendant nos discussions. Même minime. Car cela aide à ne pas revenir sur des décisions/idées qu'on a jugé mauvaises.

Conclusion

A travers ces 3 sprints, nous avons répondu aux user story des deux premières semaines, mais compte tenus du retard que nous avons accumulé à cause d'une nécessité de rework du code. Nous n'avons pas couvert la totalité des scénarios de la 3ème semaine, cependant nous avons une bonne confiance en notre code grâce à nos tests et notre CI, nous avons identifié les faiblesses existantes et les refactors à appliquer en conséquence. De même, pour ce qui est des user story de la 3ème semaine, nous savons comment implémenter ce qui nous manque, et notre architecture actuel est prête pour l'arrivé de ces "nouvelles features". Nous avons également identifié ce qui n'a pas marché dans les précédentes itérations et sauront en tenir compte pour la suite.